CODASIP JUMP THREADING

Abstract

Jump threading is a type of compilation optimization that aims to produce faster code. Codasip introduced its own implementation of jump threading in its compiler that is based on LLVM and is a part of Codasip's unique tool suite, Codasip Studio. This whitepaper describes how jump threading works in general, how Codasip jump threading pass works and how it can be used to further improve speed of a program execution. Jump threading passes of existing compilers will be used in comparison.

Introduction

The most widely used C/C++ compilers today are open source: GCC by the GNU Project and Clang by the LLVM project. The Codasip C/C++ compiler is based on LLVM. LLVM is an umbrella project that hosts a set of related low-level toolchain components (assemblers, compilers, debuggers, etc.). LLVM and its C/C++ frontend, Clang, provide a number of benefits over GCC, specifically faster compilation and lower memory usage, expressive diagnostics and modular library-based architecture that allows easy customization and addition of custom extensions in the form of new architectures, instructions, and optimizations.

One of the stronger points of GCC, however, is that jump threading pass in GCC is more powerful than the same pass in LLVM, which also has difficulties in threading jumps used in CoreMark benchmarking. To mitigate it and improve our own LLVM solution, we developed an innovative implementation of jump threading that helped us achieve significantly faster code and better CoreMark results.

General Principles of Jump Threading

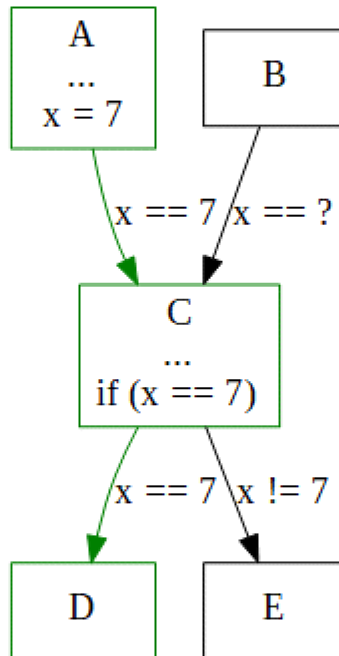
Jump threading is a compiler pass that attempts to speed up code execution by replacing conditional jumps with unconditional ones. Execution speed can be increased at the cost of increasing code size. Full definition of jump threading and its benefits is given in **References [1]**. Let's use an example to illustrate how the optimization works.

Codasip GmbH – Headquarters
Paul-Gerhardt-Allee 50
81245 Munich, Germany

Codasip s.r.o. – R&D Center
Technická 2935/23
616 00 Brno, Czech Republic

Phone: +420 519 810 190
E-mail: contact@codasip.com
Website: www.codasip.com

Graph 1 consists of five basic blocks: A, B, C, D, and E. In basic block A, variable x is set to 7.



Graph 1

When the value of x is 7, basic block C jumps to basic block D; in any other case, it jumps to basic block E.

To optimize execution of the resultant code, we need to identify and isolate any *threading opportunities* in the control flow graph. A threading opportunity is a path from start block to target block where the target block is executed when the sequence of start block and condition block is executed, regardless of program state.

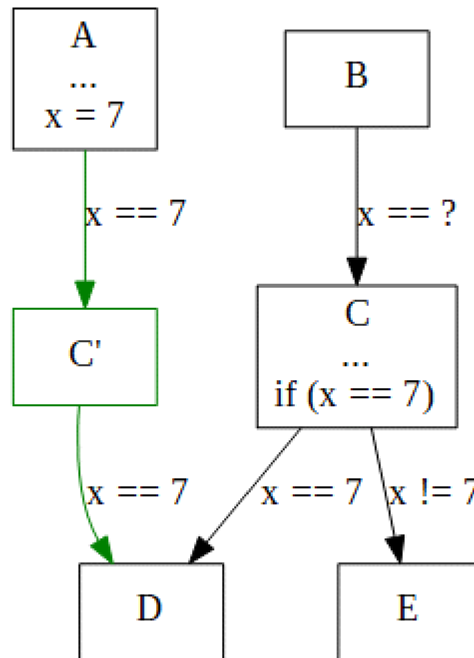
Our example contains one threading opportunity: the path A → C → D. A is a start block, C is a condition block, and D is a target block. Block D will be always executed when basic blocks A with C are executed.

Jump threading can be executed when the threading opportunity is defined. In our example, the following changes are required: First, the condition block C must be duplicated, let's name it C′. Then the destination of edge from start block to condition block must be rerouted to the duplicated block. Finally, the conditional jump of the duplicated block needs to be changed to an unconditional jump. Result: The path with unconditional jump A → C′ → D is used instead of the path with conditional jump A → C → D at the expense of duplicated basic block C′. *Graph 2* shows the changed flow.

Codasip GmbH – Headquarters
Paul-Gerhardt-Allee 50
81245 Munich, Germany

Codasip s.r.o. – R&D Center
Technická 2935/23
616 00 Brno, Czech Republic

Phone: +420 519 810 190
E-mail: contact@codasip.com
Website: www.codasip.com



Graph 2

Existing Jump Threading Implementations

As mentioned, jump threading is used in both major open source compilers, Clang/LLVM and GCC. We also examined LibFIRM+, the latest improved implementation of LibFIRM mentioned in **References [1]** (for the purpose of this paper, all instances of LibFIRM actually refer to LibFIRM+).

Clang (LLVM)

Clang iterates through all basic blocks, one by one. For each basic block, Clang evaluates its jump condition. When the condition can be evaluated, Clang identifies a target block with the resulting value, then continues by analyzing predecessors. Jump threading is performed only when it is certain that no irreducible graph will be created. Thus, Clang supports only threading opportunities of length 3, and is very conservative compared to other solutions.

GCC

GCC first analyses the control flow graph and collects all threading paths. Then it sorts them and selects the best candidates for jump threading. The compiler also attempts to avoid irreducible graphs, but the rules are not as restrictive as in Clang.

LibFIRM+

LibFIRM, too, begins by finding all threading opportunities, and represents them in a compact way. The representation is designed so that even infinite threading opportunities can be described in a finite way.

Codasip Jump Threading

Out of the existing implementations, LibFIRM analysis of threading opportunities is the most optimal type. In practice, however, adopting the algorithm for LLVM is not easy because the LibFIRM representation is very different from LLVM IR and some parts would not work. For this reason, Codasip jump threading had to be created as a completely new solution, inspired by LibFIRM but created specifically for LLVM IR.

LLVM IR

To illustrate LLVM IR (intermediate representation), we will use code generated for an example program, see Appendix A: Example program.

```
define dso_local i32 @test(i32 %x) local_unnamed_addr #0 {
entry:
   %tobool = icmp eq i32 %x, 0
   br i1 %tobool, label %if.end, label %if.then

if.then:                                   ; preds = %entry
   %call = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
    ([13 x i8], [13 x i8]* @.str, i32 0, i32 0))
   br label %if.end

if.end:                                    ; preds = %entry, %if.then
   %x.addr.0 = phi i32 [ 4, %if.then ], [ %x, %entry ]
   %cmp = icmp slt i32 %x.addr.0, 3
   br i1 %cmp, label %if.then1, label %if.end3

if.then1:                                  ; preds = %if.end
   %call2 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
    ([13 x i8], [13 x i8]* @.str, i32 0, i32 0))
   br label %if.end3

if.end3:                                   ; preds = %if.then1, %if.end
   ret i32 %x.addr.0
}
```

The LLVM IR code above corresponds to the `test` function of the example program. It consists of five basic blocks, the most important of which is `if.end`. This block contains a PHI node showing that value of x is set to 4 when the executed predecessor was basic block `if.then`. We use this information in evaluation of conditional jump condition of basic block `if.end`; the resulting threading opportunity will be `if.then` → `if.end` → `if.end3`.

Codasip Jump Threading Structures

Structures are introduced as an auxiliary concept, as it is not convenient to work directly with LLVM IR. Also, they help take a similar approach as GCC or LibFIRM do: perform a thorough analysis first, then apply results of the analysis in one step.

The main structure is a **JTGraph**. JTGraph consists of JTBlocks and JTEdges. It contains mapping of LLVM basic blocks to corresponding JTBlocks, and it stores information that helps simplify the process of merging JTGraphs. Primarily, a JTGraph represents LLVM function; additionally, it represents threading opportunities.

A JTBlock is built over a LLVM basic block and has a pointer to it. If the JTBlock is duplicated, it also contains a pointer to the original JTBlock. Finally, it contains information about the predecessor and successor JTEdges.

A JTEdge represents a single oriented edge between two JTBlocks. It contains information about JTEdge type, which can be an unconditional jump, conditional jump, switch, or indirect jump. It also defines the source JTBlock, destination JTBlock, and value for the jump.

Codasip Jump Threading Algorithm

The algorithm of Codasip jump threading can be broken down into the following steps, explained in detail in the subsequent chapters:

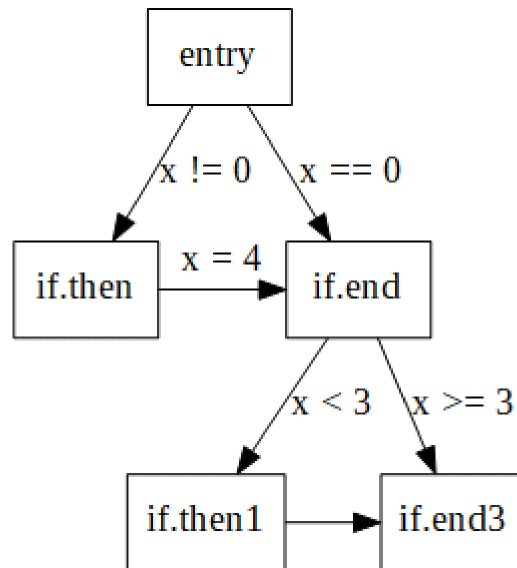
1. Create main JTGraph over LLVM function.
2. Find threadable graphs.
3. Merge threadable graphs.

   If no threadable graphs were found, or all found graphs have been merged so that no jump of LLVM function can be threaded, algorithm ends at this point.

4. Merge threadable graphs with the same prefix.
5. Attach threadable graphs to the main JTGraph. (All attached blocks need to be duplicated.)
6. Identify reachable blocks in the main JTGraph.
7. Delete predecessor edges leading to unreachable blocks.
8. Clone LLVM basic blocks to match the duplicated JTBlocks.
9. Fix terminators of the cloned LLVM basic blocks.
10. Rebuild PHI nodes of all LLVM basic blocks.
11. Add duplicated blocks into the LLVM function.
12. Remove dead PHI nodes and unreachable blocks.

1. Create Main JTGraph Over LLVM Function

This is a straightforward step: Iterate through all basic blocks of LLVM function and create a JTBlock over each LLVM basic block, then iterate again to fill JTEdges in accordance with the LLVM IR. *Graph 3* shows the initial JTGraph of our example program.



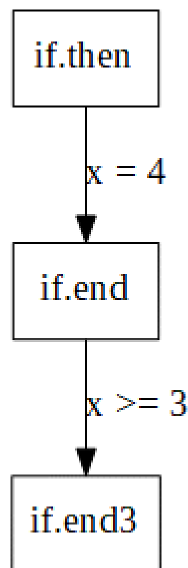
Graph 3

2. Find Threadable Graphs

To find all threadable graphs, a comprehensive analysis of the main JTGraph is run. The result is a vector of JTGraphs. Each conditional jump condition is evaluated backwards in the process; the aim is to model the backward control flow for evaluation of condition. All JTBlocks that pass the backward evaluation are remembered and their cost is summed. If the cost is higher than a threshold set in `-codasip-jump-threading-cost-threshold` argument, the evaluation stops, and no threading opportunity of remaining evaluation is found. Evaluation stops when the result is a constant, or a constant address (in case of indirect jumps). Evaluation of condition multiplies on PHI nodes, starting a separate evaluation instance on each predecessor of the PHI node.

Let's take the condition expression `%cmp` of a conditional jump in our example. It is not a constant, so the process continues evaluating `%cmp` into `icmp slt i32 %x.addr.0, 3`. Now `%x.addr.0` is still not a constant, but it is a PHI node, so the evaluation splits.

Path 1: First, by evaluating `%x.addr.0` with 4, we get `icmp slt i32 4, 3`, which is 0 and it is a constant. Evaluation of this path is complete. Now we summarize the backward control flow path (this is done with the cost during the process, but it is simpler to describe separately): The first JTBlock is determined by the variable `%cmp` – it is the block `if.end`. Variable `%x.addr.0` is inside the same block as variable `%cmp`, preceding it, so the same JTBlock is not added twice. The last JTBlock `if.then` comes from PHI node information. *Graph 4* shows the resulting JTGraph of this threading opportunity.



Graph 4

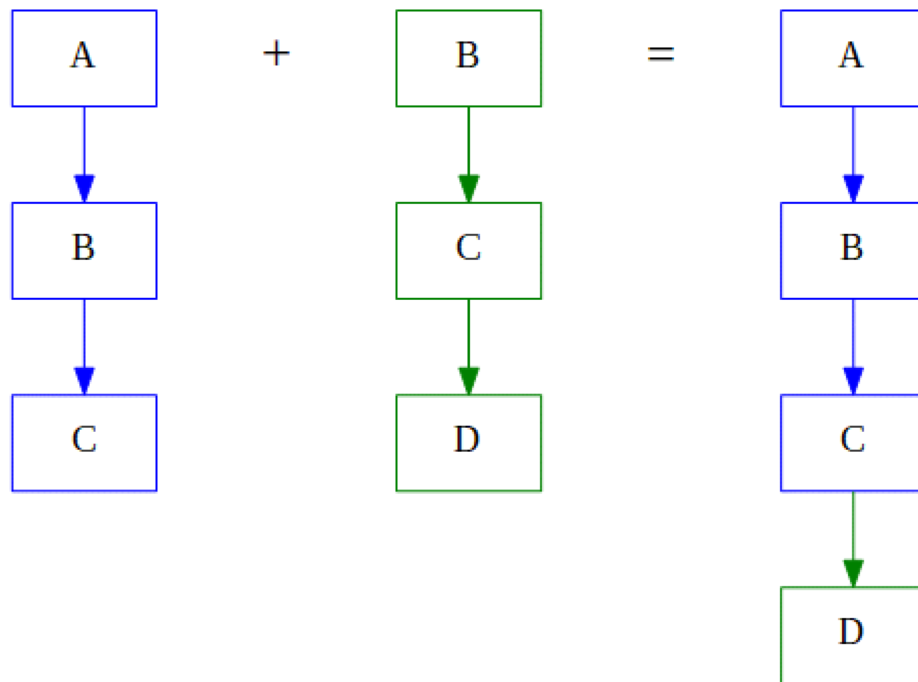
Path 2: Evaluation continues with the second PHI node variant of `%x.addr.0`. If evaluated with `%x` into `icmp slt i32 %x, 3`, it is not possible to continue evaluating in compile time, which means that this path does not lead into any new threading opportunity, and it is discarded. Paths leading to a higher cost than the specified threshold are also discarded.

Result: We found one threading opportunity for this condition, and as there is no other evaluation of conditional jump condition that would lead to a new threading opportunity, we can conclude that there is only one threading opportunity in total for the LLVM function.

3. Merge Threadable Graphs

Iteratively, merge the found threadable graphs to reduce their number, if possible. Sometimes there are multiple threading opportunities found that overlap, and this pass aims to lower their amount by merging them into bigger ones, which helps reduce the number of duplicated LLVM blocks and improves performance on LLVM.

This step does not affect the main example we use, so we will present a new separate example to illustrate the merging in *Graph 5*.



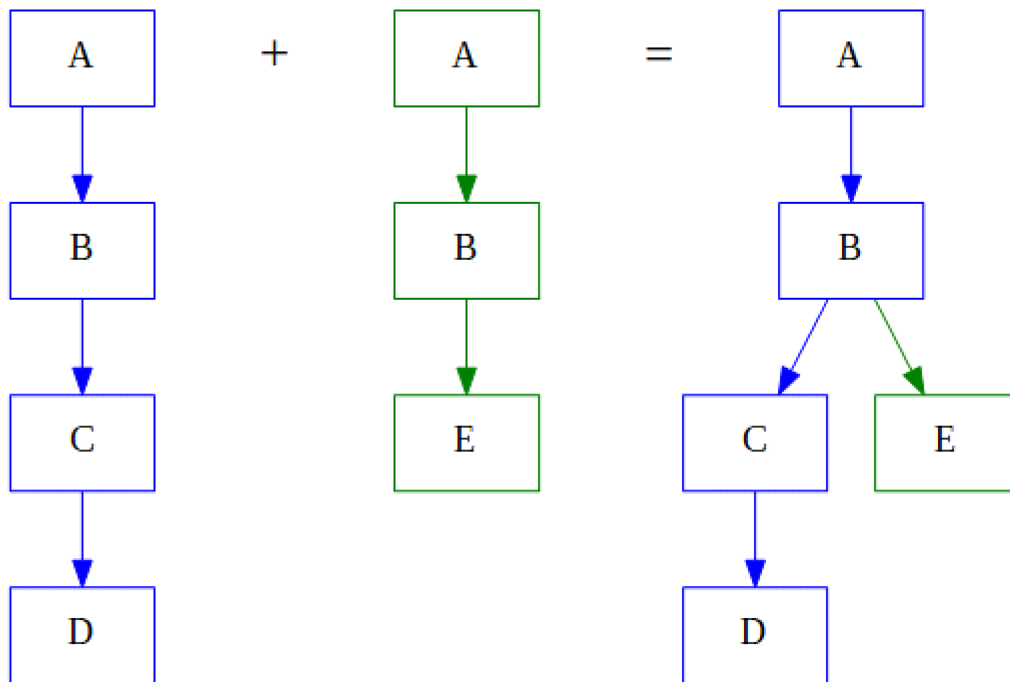
Graph 5

In the blue graph, block C is always executed after block B is executed. Similarly, the green graph shows that block D is executed after both block B and block C are executed. The two graphs can therefore be merged into one longer threading opportunity. The resulting graph tells us that when block A and block B are executed, then both block C and block D are executed, too.

4. Merge Threadable Graphs with the Same Prefix

In the next step, the algorithm merges threadable graphs that share the same start block and first condition block. Merging is done on last condition block of the prefix. This step is essential as it would be otherwise impossible to attach the threading opportunities to the main JTGraph in the attaching phase.

Again, this phase does not have any effect on the main example, so we will use another individual example in *Graph 6*.

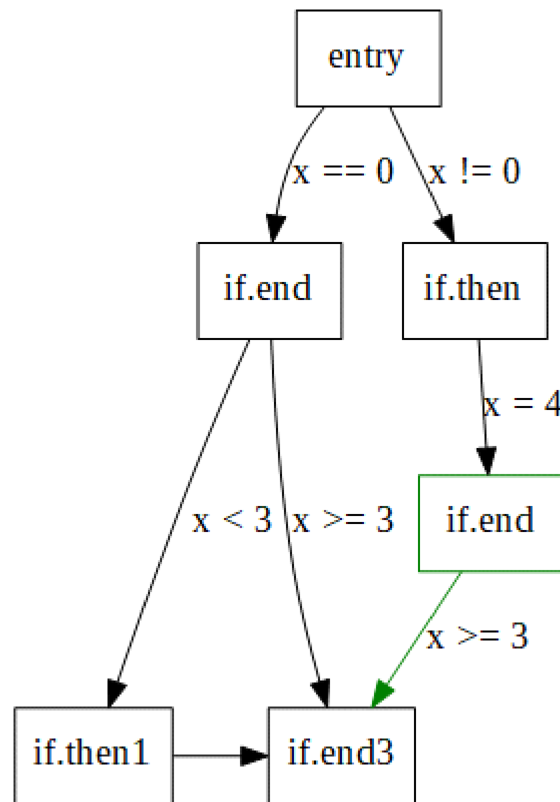


Graph 6

The two threadable graphs, blue and green, share the same prefix A → B. The green graph must be merged on the last condition block of the blue graph prefix, which is B. As a result, the whole blue graph is preserved, and the blocks and edges from the green graph that are not part of the shared prefix are attached to the condition block B of the blue graph.

5. Attach Threadable Graphs to the Main JTGraph

Threadable graphs must be now attached to the main JTGraph. The start block and its successor edge of the threaded graph are used as a key for searching the main JTGraph. The first conditional block is set as a destination block into the found edge; the destination block vector of predecessor edges is updated as well at this point. The rest is already valid. *Graph 7* shows the result.



Graph 7

Newly attached objects are green. All newly attached JTBlocks need to be duplicated; in this case, only LLVM basic block `if.end` must be duplicated.

6. Identify Reachable Blocks in the Main JTGraph

Reachable blocks are collected by recursive traversing of the main JTGraph from the entry block. In the presented example, all JTBlocks are reachable.

7. Delete Predecessor Edges Leading to Unreachable Blocks

The algorithm iterates through all blocks of the main JTGraph, deleting all predecessor edges that lead to unreachable blocks. This step does not apply to the main example, either, as it contains no unreachable blocks.

8. Clone LLVM Basic Blocks to Match the Duplicated JTBlocks

LLVM basic blocks are now cloned to correspond to duplicated JTBlocks. Cloning is done by using `CloneBasicBlock` and `RemapInstruction` on each duplicated block.

In our example, there is only one block that must be duplicated: the green block `if.end`. LLVM IR of a newly duplicated LLVM basic block follows.

```
if.end.jt0:                                  ; Error: Block without parent!
   %x.addr.0 = phi i32 [ 4, %if.then ], [ %x, %entry ]
   %cmp = icmp slt i32 %x.addr.0, 3
   br i1 %cmp, label %if.then1, label %if.end3
```

9. Fix Terminators of the Cloned LLVM Basic Blocks

Terminators of the duplicated LLVM basic blocks must be updated to match terminators of the corresponding JTBlocks. A duplicated JTBlock usually has an unconditional jump as a terminator, but the original LLVM block has a conditional jump. This phase is the key part of Codasip jump threading.

In our example, terminator of the basic block `if.end.jt0` needs to be fixed to match the duplicated JTBlock.

```
if.end.jt0:                              ; Error: Block without parent!
   %x.addr.0 = phi i32 [ 4, %if.then ], [ %x, %entry ]
   %cmp = icmp slt i32 %x.addr.0, 3
   br label %if.end3
```

10. Rebuild PHI Nodes of All LLVM Basic Blocks

Algorithm for rebuilding PHI nodes was inspired by the algorithm described in **References [1]**, with some minor changes related to fixing bugs, found by using real world programs, and adjusting to LLVM IR structure.

This phase computes and uses dominating definitions to compute new or correct old PHI nodes for each instruction variable.

The following LLVM IR contains two changes. First, variable `%x.addr.0` of duplicated basic block was renamed to `%x.addr.0.jt1`. Second, PHI node predecessors were corrected to match the JTBlock predecessors.

```
if.end.jt0:                              ; Error: Block without parent!
   %x.addr.0.jt1 = phi i32 [ 4, %if.then ]
   %cmp = icmp slt i32 %x.addr.0.jt1, 3
   br label %if.end3
```

11. Add Duplicated Blocks into the LLVM Function

Duplicated blocks cannot be placed into the LLVM function immediately after creation because LLVM renames variables of duplicate names. This would lead to broken PHI nodes and other errors. For this reason, duplicated blocks must be added to LLVM function *after* variables and PHI nodes are correctly processed.

The following example presents LLVM IR after adding the duplicated basic block into LLVM function. Note that the variable `%cmp` was automatically renamed to `%cmp6` by LLVM. This explains why this step cannot be performed sooner in the process. Variable `%cmp` does not have any real use after the terminator was fixed, so the automatic renaming does not break anything, and the dead code can be optimized later.

```
if.end.jt0:                                       ; preds = %if.then
   %x.addr.0.jt1 = phi i32 [ 4, %if.then ]
   %cmp6 = icmp slt i32 %x.addr.0.jt1, 3
   br label %if.end3
}
```

12. Remove Dead PHI Nodes and Unreachable Blocks

Finally, the algorithm removes unreachable blocks from LLVM function and deletes any dead PHI nodes to produce clean LLVM IR.

This step does affect our example. The following code shows the resultant LLVM IR after completing the Codasip jump threading algorithm.

```
; Function Attrs: nounwind

define dso_local i32 @test(i32 %x) local_unnamed_addr #0 {
entry:
   %tobool = icmp eq i32 %x, 0
   br i1 %tobool, label %if.end, label %if.then

if.then:                                     ; preds = %entry
   %call = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
    ([13 x i8], [13 x i8]* @.str, i32 0, i32 0))
   br label %if.end.jt0

if.end:                                      ; preds = %entry
   %x.addr.0 = phi i32 [ %x, %entry ]
   %cmp = icmp slt i32 %x.addr.0, 3
   br i1 %cmp, label %if.then1, label %if.end3

if.then1:                                    ; preds = %if.end
   %call2 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
    ([13 x i8], [13 x i8]* @.str, i32 0, i32 0))
   br label %if.end3

if.end3:                                     ; preds = %if.end.jt0,
    %if.then1, %if.end
   %x.addr.0.jt0 = phi i32 [ %x.addr.0, %if.end ], [ %x.addr.0,
    %if.then1 ], [ %x.addr.0.jt1, %if.end.jt0 ]
   ret i32 %x.addr.0.jt0

if.end.jt0:                                  ; preds = %if.then
   %x.addr.0.jt1 = phi i32 [ 4, %if.then ]
   %cmp6 = icmp slt i32 %x.addr.0.jt1, 3
   br label %if.end3
}
```

Usage

Codasip jump threading can be turned on by adding these arguments:

```
--mllvm -use-codasip-jump-threading
```

Basic usage is:

```
clang -O3 --mllvm -use-codasip-jump-threading
```

The pass can be turned on only when optimize most (`-O3`) is present and optimize for code size is not present (`-Os, -Oz`).

The following argument sets the maximum cost of analysis for threading opportunities:

```
-codasip-jump-threading-cost-threshold=30.
```

Advanced usage then is:

```
clang -O3 --mllvm -use-codasip-jump-threading --mllvm -codasip-
    jump-threading-cost-threshold=30.
```

The above command runs Codasip jump threading with cost set on 30, which is the default value.

Results

When comparing CoreMark benchmarks, Codasip jump threading proved to bring significant improvement against out-of-the-box LLVM jump threading:

Without LTO (`-flto`), CoreMark benchmarks were **13 %** better than LLVM when cost threshold of Codasip jump threading was set to 30 (default value), and **17.6 %** better when cost threshold was set to 120.

With LTO (`-flto`), the improvement was **16.5 %** for any cost threshold value. Cost threshold did not make any difference in this case because LTO is a very aggressive optimization. As such, it removes any threading opportunities when applied, so Codasip jump threading optimization does not have any effect on the result.

Conclusion

This paper described the general idea of jump threading optimization, and briefly introduced its existing implementations in major compilers. Codasip jump threading algorithm was then described in detail and illustrated on a simple example. We also explained how to use the feature. Finally, measured results were presented, showing noticeably improved speed.

In the future, it is desirable to further upgrade the Codasip solution, either by improving the presented Codasip jump threading pass, or by identifying another pass that can be efficiently improved. Further work on improving the existing solution can include determining a better pass order in LLVM, turning off passes that interfere with output of Codasip jump threading, and use of profiling information, namely PGO, to thread only jumps that are heavily used, as opposed to processing all jumps. Also, the first merging phase can be improved by including smaller threading opportunities that were used in the merging process and then discarded; this would lead to more duplicated basic blocks and bigger size, but the resultant code should be faster.

References

[1] PRIESNER, Joachim. *Generalized Jump Threading in LibFIRM* [online]. Karlsruhe, Germany, 2017. https://pp.ipd.kit.edu/uploads/publikationen/priesner17masterarbeit.pdf. Master Thesis. Karlsruhe Institute of Technology.

Appendix A: Example program

This example program has been used in the detailed description of the Codasip jump threading algorithm.

```
int test(int x)
{
   if (x) {
      printf("hello world!");
      x = 4;
   }
   if (x < 3) {
      printf("hello world!");
   }
   return x;
}

int main(int argc, char* argv[])
{
   return test(argc);
}
```